PCT

WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification [6] : G06F 11/14, 9/46, 17/30 — A1

(11) International Publication Number: WO 97/04391

(43) International Publication Date: 6 February 1997 (06.02.97)

(21) International Application Number: PCT/US96/11903

(22) International Filing Date: 18 July 1996 (18.07.96)

(30) Priority Data:
60/001,245 — 20 July 1995 (20.07.95) — US

(60) Parent Application or Grant
(63) Related by Continuation
US — 60/001,245 (CIP)
Filed on — 20 July 1995 (20.07.95)

(71) Applicant (*for all designated States except US*): NOVELL, INC. [US/US]; 1555 North Technology Way, Orem, UT 84057 (US).

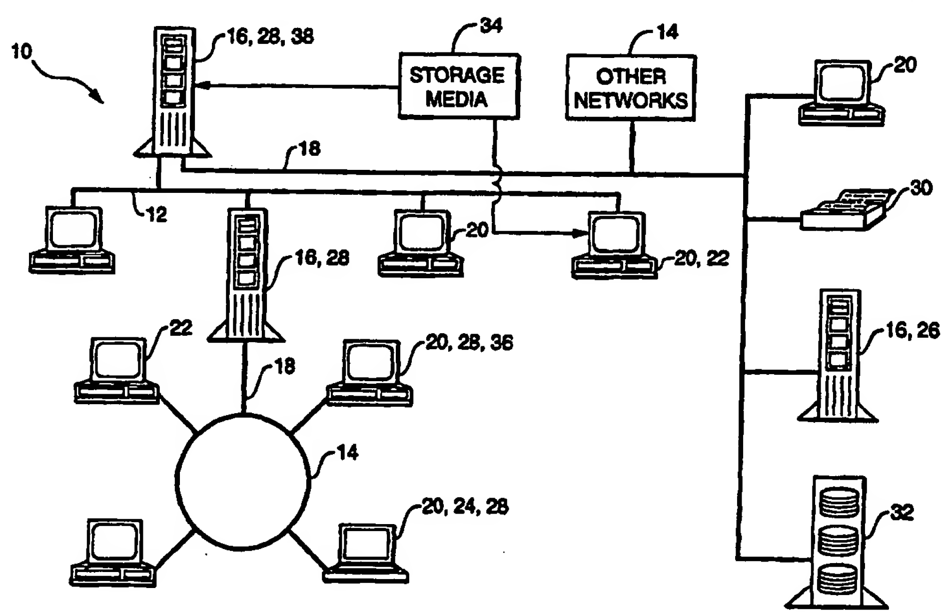
(72) Inventors; and
(75) Inventors/Applicants (*for US only*): FALLS, Patrick, T. [GB/GB]; Meadlands, Broad Layings, Woolton Hill, Newbury, Berkshire RG15 9TT (GB). COLLINS, Brian, J. [GB/GB]; 30 High Drive, New Malden, Surrey KT3 3UG (GB). DRAPER, Stephen, P., W. [GB/GB]; 123 Pack Lane, Basingstoke, Hampshire RG22 5HL (GB).

(74) Agents: OGILVIE, John, W., L. et al.; Computer Law++, Suite 550, 8 East Broadway, Salt Lake City, UT 84111 (US).

(81) Designated States: AL, AM, AT, AU, AZ, BB, BG, BR, BY, CA, CH, CN, CZ, DE, DK, EE, ES, FI, GB, GE, HU, IL, IS, JP, KE, KG, KP, KR, KZ, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, TJ, TM, TR, TT, UA, UG, US, UZ, VN, ARIPO patent (KE, LS, MW, SD, SZ, UG), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, ML, MR, NE, SN, TD, TG).

Published
With international search report.

(54) Title: TRANSACTION LOG MANAGEMENT IN A DISCONNECTABLE COMPUTER AND NETWORK



(57) Abstract

A method and apparatus are disclosed for managing a transaction log which contains updates representing operations performed on a database replica in a network of disconnectable computers. The invention provides for compression of the log by the identification and removal of redundant updates. Log compression removes apparent inconsistencies between operations performed on disconnected computers, reduces storage requirements on each computer, and speeds up transaction synchronization when the computers are reconnected. The invention also provides for restoration of prior versions of database objects using the log.

TITLE

TRANSACTION LOG MANAGEMENT IN A DISCONNECTABLE COMPUTER AND NETWORK

INVENTORS

STEPHEN P.W. DRAPER, BRIAN J. COLLINS, AND PATRICK T. FALLS

FIELD OF THE INVENTION

The present invention relates to the management of transaction logs which contain updates representing operations performed on separated disconnectable computers, and more particularly to log compression that is suitable for use with transaction synchronization and with the handling of clashes that may arise during such synchronization.

TECHNICAL BACKGROUND OF THE INVENTION

"Disconnectable" computers are connected to one another only sporadically or at intervals. Familiar examples include "mobile-link" portable computers which are connectable to a computer network by a wireless links and separate server computers in a wide-area network (WAN) or other network. Disconnectable computers can be operated either while connected to one another or while disconnected. During disconnected operation, each computer has its own copy of selected files (or other structures) that may be needed by a user. Use of the selected items may be either direct, as with a document to be edited, or indirect, as with icon files to be displayed in a user interface.

Unfortunately, certain operations performed on the selected item copies may not be compatible or consistent with one another. For instance, one user may modify a file on one computer and another user may delete the "same" file from the other computer. A "synchronization" process may be performed after the computers are reconnected. At a minimum, synchronization attempts to propagate operations performed on one computer to the other computer so that copies of items are consistent with one another.

for synchronization is relatively slow, as many modem or WAN links are.

Moreover, in some conventional approaches potentially conflicting changes to a given set of data are handled by simply applying the most recent change and discarding the others. In other conventional systems, users must resolve conflicts with little or no assistance from the system. This can be both tedious and error-prone.

It is well-known in the database arts to maintain a log of transactions. However, conventional disconnectable systems are not traditional database systems. Conventional disconnectable systems lack transaction logs which can be used to identify and then modify or remove certain apparently inconsistent operations to improve the synchronization process. Conventional systems provide no way to compress transaction logs based on the semantics of the logged update operations. Conventional systems also lack a way to use such transaction logs to recreate earlier versions of database objects.

Thus, it would be an advancement in the art to provide a system and method for compressing a log of transactions performed on disconnectable computers.

It would be a further advancement to provide such a system and method which are suited for use with systems and methods for transaction synchronization.

It would also be an advancement to provide such a system and method which are not limited to file system operations but can instead be extended to support a variety of database objects.

Such a system and method are disclosed and claimed herein.

BRIEF SUMMARY OF THE INVENTION

The present invention provides systems and methods for managing a transaction log which represents a sequence of transactions in a network of connectable computers. Each transaction contains at least one update targeting an object in a replica of a distributed target database. The replicas reside on separate computers in the network. In one embodiment the network includes a server computer and a client computer

The present log management invention is suitable for use with various transaction synchronization systems and methods. According to one such, synchronization of the database replicas is performed after the computers are reconnected and includes a "merging out" step, a "merging in" step, and one or more clash handling steps. During the merging out step, operations performed on a first computer are transmitted to a second computer and applied to a replica on the second computer. During the merging in step, operations performed on the second computer are transmitted to the first computer and applied to the first computer's replica.

Some of the clash handling steps detect transient or persistent clashes, while other steps recover from at least some of those clashes. Persistent clashes may occur in the form of unique key clashes, incompatible manipulation clashes, file content clashes, permission clashes, or clashes between the distributed database and an external structure. Recovery may involve insertion of an update before or after a clashing update, alteration of the order in which updates occur, consolidation of two updates into one update, and/or creation of a recovery item. Log compression may be performed as part of clash handling, in preparation for merging, or separately from those procedures.

Transaction synchronization and clash handling are further described in commonly owned copending applications entitled TRANSACTION SYNCHRONIZATION IN A DISCONNECTABLE COMPUTER AND NETWORK and TRANSACTION CLASH MANAGEMENT IN A DISCONNECTABLE COMPUTER AND NETWORK, filed the same day and having the same inventors as the present application.

The features and advantages of the present invention will become more fully apparent through the following description and appended claims taken in conjunction with the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

To illustrate the manner in which the advantages and features of the invention are obtained, a more particular description of the invention summarized above will be rendered by reference to the appended drawings. Understanding that

combination thereof. The servers 16 and the network clients 20 may be configured by those of skill in the art in a wide variety of ways to operate according to the present invention.

The network clients 20 include personal computers 22, laptops 24, and workstations 26. The servers 16 and the network clients 20 are collectively denoted herein as computers 28. Suitable computers 28 also include palmtops, notebooks, personal digital assistants, desktop, tower, micro-, mini-, and mainframe computers. The signal lines 18 may include twisted pair, coaxial, or optical fiber cables, telephone lines, satellites, microwave relays, modulated AC power lines, and other data transmission means known to those of skill in the art.

In addition to the computers 28, a printer 30 and an array of disks 32 are also attached to the illustrated network 10. Although particular individual and network computer systems and components are shown, those of skill in the art will appreciate that the present invention also works with a variety of other networks and computers.

At least some of the computers 28 are capable of using floppy drives, tape drives, optical drives or other means to read a storage medium 34. A suitable storage medium 34 includes a magnetic, optical, or other computer-readable storage device having a specific physical substrate configuration. Suitable storage devices include floppy disks, hard disks, tape, CD-ROMs, PROMs, RAM, and other computer system storage devices. The substrate configuration represents data and instructions which cause the computer system to operate in a specific and predefined manner as described herein. Thus, the medium 34 tangibly embodies a program, functions, and/or instructions that are executable by at least two of the computers 28 to perform log management steps of the present invention substantially as described herein.

With reference to Figure 2, at least two of the computers 28 are disconnectable computers 40 configured according to the present invention. Each disconnectable computer 40 includes a database manager 42 which provides a location-independent interface to a distributed hierarchical target database embodied in convergently consistent replicas 56. Suitable

requests of the database manager 42, which dispatches each request to an appropriate agent 44.

Each agent 44 embodies semantic knowledge of an aspect or set of objects in the distributed target database. Under this modular approach, new agents 44 can be added to support new distributed services. For instance, assumptions and optimizations based on the semantics of the hierarchy of the NetWare File System are embedded in a Hierarchy Agent, while corresponding information about file semantics are embedded in a File Agent. In one embodiment, such semantic information is captured in files defining a schema 84 (Figure 3) for use by agents 44.

The schema 84 includes a set of "attribute syntax" definitions, a set of "attribute" definitions, and a set of "object class" (also known as "class") definitions. Each attribute syntax in the schema 84 is specified by an attribute syntax name and the kind and/or range of values that can be assigned to attributes of the given attribute syntax type. Attribute syntaxes thus correspond roughly to data types such as integer, float, string, or Boolean in conventional programming languages.

Each attribute in the schema 84 has certain information associated with it. Each attribute has an attribute name and an attribute syntax type. The attribute name identifies the attribute, while the attribute syntax limits the values that are assumed by the attribute.

Each object class in the schema 84 also has certain information associated with it. Each class has a name which identifies this class, a set of super classes that identifies the other classes from which this class inherits attributes, and a set of containment classes that identifies the classes permitted to contain instances of this class.

An object is an instance of an object class. The target database contains objects that are defined according to the schema 84 and the particulars of the network 10. Some of these objects may represent resources of the network 10. The target database is a "hierarchical" database because the objects in the database are connected in a hierarchical tree structure. Objects in the tree that can contain other objects are called

```
ATTRIBUTE
{
     ndr_dodb_dummy_key  dummy_key
     PROPERTY            NDR_OS_ATTR_FLAG_SIBLING_KEY
     COMPARISON          ndr_dodb_dummy_key_compare
     VALIDATION          ndr_dodb_dummy_key_validate;
     ha_volume_id        next_free_volume_id;
}
}
```

A file or directory name can be 12 (2-byte) characters long:

```
CONSTANT  HA_FILENAME_MAX = 24;
DATATYPE  ha_filename          STRING HA_FILENAME_MAX;
```

The ha_file_or_dir_id is a compound unique key embracing the file or directory ID that is allocated by the server, as well as the server-generated volume number. The latter is passed as a byte from class 87 NetWare Core Protocols from which it is read directly into vol (declared as a byte below). Elsewhere in the code the type ndr_host_volume_id (a UINT16) is used for the same value.

```
DATATYPE    ha_file_or_dir_id
{
    ULONG            file_or_dir;
    ha_volume_id     vol;
}
```

Files and directories have many shared attributes, the most important being the file name. This must be unique for any parent directory.

```
CLASS      ha_file_or_dir
{
     PARENT           ha_directory;
     SUPERCLASS       ndr_dodb_object_header;
     ATTRIBUTE
     {
          ha_filename          filename
          PROPERTY             NDR_OS_ATTR_FLAG_SIBLING_KEY |
                               NDR_OS_ATTR_FLAG_IS_DOS_FILENAME;
          ha_file_or_dir_id    id
          PROPERTY             NDR_OS_ATTR_FLAG_GLOBAL_KEY |
                               NDR_OS_ATTR_FLAG_UNREPLICATED
          GROUP                file_or_dir_id_group;
          ULONG                attributes;
          SHORT                creation_date;
```

```
                                   NDR_OS_ATTR_FLAG_UNREPLICATED;
    }
}
```

The root directory must appear at the top of the hierarchy below the volume. Its name is not used; the volume name is used instead. This is the top of the replication hierarchy and therefore is the top level RSC in this hierarchy:

```
CLASS      ha_root_directory
{
     SUPERCLASS      ha_directory;
     PARENT          ha_volume;
     PROPERTY        NDR_OS_CLASS_FLAG_DEFINE_REPLICAS |
                     NDR_OS_CLASS_FLAG_HAS_RSC;
}
```

In one embodiment, schemas such as the schema 84 are defined in a source code format and then compiled to generate C language header files and tables. The named source file is read as a stream of lexical tokens and parsed using a recursive descent parser for a simple LL(1) syntax. Parsing an INCLUDE statement causes the included file to be read at that point. Once a full parse tree has been built (using binary nodes), the tree is walked to check for naming completeness. The tree is next walked in three passes to generate C header (.H) files for each included schema file. The header generation passes also compute information (sizes, offsets, and so forth) about the schema which is stored in Id nodes in the tree. Finally, the complete tree is walked in multiple passes to generate the schema table C source file, which is then ready for compiling and linking into an agent's executable program.

Each disconnectable computer 40 also includes a replica manager 46 which initiates and tracks location-specific updates as necessary in response to database manager 42 requests. The

with extended NetWare Core Protocol calls and provides functionality according to the following interface:

rpc_init()	Initialize RPC subsystem
rpc_shutdown()	Shutdown RPC subsystem
rpc_execute()	Execute request at single location
rpc_ping()	Ping a location (testing)
rpc_claim_next_execute()	Wait until the next rpc_execute() is guaranteed to be used by this thread
rpc_free_next_execute()	Allow others to use rpc_execute()

Those of skill in the art will appreciate that other remote procedure call mechanisms may also be employed according to the present invention. Suitable network connections 52 may be established using packet-based, serial, internet-compatible, local area, metropolitan area, wide area, and wireless network transmission systems and methods.

Figures 2 and 3 illustrate one embodiment of the replica manager 46 of the present invention. A replica distributor 70 insulates the database manager 42 from the complexities caused by having database entries stored in replicas 56 on multiple computers 40 while still allowing the database manager 42 to efficiently access and manipulate individual database objects, variables, and/or records. A replica processor 72 maintains information about the location and status of each replica 56 and ensures that the replicas 56 tend to converge.

A consistency distributor 74 and a consistency processor 76 cooperate to maintain convergent and transactional consistency of the database replicas 56. The major processes used include an update process which determines how transaction updates are applied, an asynchronous synchronization process that asynchronously synchronizes other locations in a location set, a synchronous synchronization process that synchronously

update location. Each group of updates associated with a single transaction have a processor transaction identifier ("PTID") containing the location identifier of the update location and a transaction sequence number. The transaction sequence number is preferably monotonically consecutively increasing for all completed transactions at a given location, even across computer 28 restarts, so that other locations receiving updates can detect missed updates.

The PTID is included in update details written to an update log by an object processor 86. An update log (sometimes called an "update stream") is a chronological record of operations on the database replica 56. Although it may be prudent to keep a copy of an update log on a non-volatile storage device, this is not required. The operations will vary according to the nature of the database, but typical operations include adding objects, removing objects, modifying the values associated with an object attribute, modifying the attributes associated with an object, and moving objects relative to one another.

The PTID is also included as an attribute of each target database object to reflect the latest modification of the object. In one embodiment, the PTID is also used to create a unique (within the target database) unique object identifier ("UOID") when a target database object is first created.

A target database object may contain attributes that can be independently updated. For instance, one user may set an archive attribute on a file while a second user independently renames the file. In such situations, an object schema 84 may define attribute groups. A separate PTID is maintained in the

log before they are transmitted to other locations, thereby avoiding use of the same transaction sequence number for different transactions in the event of a crash.

The SyncUpdate requests are received by other locations in the same location set and applied to their in-memory transaction logs by their respective consistency processors 76. Each consistency processor 76 only applies SyncUpdate transactions which have sequence numbers that correspond to the next sequence number for the specified location.

The consistency processor 76 can determine if it has missed updates or received them out of order by examining the PTID. If updates are missed, the PTID of the last transaction properly received is sent to the consistency distributor 74 that sent out the updates, which then arranges to send the missing updates to whichever consistency processors 76 need them.

Acknowledged requests using threads or a similar mechanism can be used in place of unacknowledged requests sent by non-central locations. Non-central locations (those not holding a master replica 56) only need to synchronize with one location and thus only require a small number of threads. To promote scalability, however, central locations preferably use unacknowledged broadcasts to efficiently transmit their SyncUpdate requests.

The asynchronous synchronization process causes SyncUpdate requests to be batched to minimize network transfers. However, the cost paid is timeliness. Accordingly, a synchronous synchronization process according to the present invention may be utilized to selectively speed up synchronization. The

Merging can also happen when two sets of computers become connected, such as when a router starts.

Merging occurs when two replicas 56 are resynchronized after the computers 28 on which the replicas 56 reside are reconnected following a period of disconnection. Either or both of the computers 28 may have been shut down during the disconnection. A set of updates are "merged atomically" if they are merged transactionally on an all-or-nothing basis. A distributed database is "centrally synchronized" if one computer 28, sometimes denoted the "central server," carries a "master replica" with which all merges are performed.

Portions of the master replica or portions of another replica 56 may be "shadowed" during a merge. A shadow replica, sometimes called a "shadow database", is a temporary copy of at least a portion of the database. The shadow database is used as a workspace until it can be determined whether changes made in the workspace are consistent and thus can all be made in the shadowed replica, or are inconsistent and so must all be discarded. The shadow database uses an "orthogonal name space." That is, names used in the shadow database follow a naming convention which guarantees that they will never be confused with names in the shadowed database.

A "state-based" approach to merging compares the final state of two replicas 56 and modifies one or both replicas 56 to make corresponding values equal. A "log-based" or "transaction-based" approach to merging incrementally applies successive updates made on a first computer 28 to the replica 56 stored on a second computer 28, and then repeats the process with the first computer's replica 56 and the second computer's

either by a background process or on demand triggered by file access. This reduces the time required for merging and promotes satisfaction of the two performance goals identified above. In embodiments tailored to "slow" links, merging is preferably on-going to take advantage of whatever bandwidth is available without substantially degrading the perceived performance of other processes running on the disconnectable computers.

In embodiments employing an update log, the log is preferably compressed prior to merging. Compression reduces the number of operations stored in the log. Compression may involve removing updates from the log, altering the parameters associated with an operation in a given update, and/or changing the order in which updates are stored in the log.

In one embodiment, all Object Database calls come through the consistency distributor 74, which manages distributed transaction processing and maintains consistency between locations. Almost all calls from a location distributor 78 are made via the consistency distributor 74 because the consistency distributor 74 supports a consistent view of the locations and the database replicas 56 on them.

The consistency distributor 74 and an object distributor 82 support multiple concurrent transactions. This is needed internally to allow background threads to be concurrently executing synchronization updates. It could also be used to support multiple concurrent gateway users. In an alternative embodiment, multiple concurrent transactions on the same session is supported through the consistency distributor 74.

```
                                      Get the effective access rights
                                      for the current session and
                                      object
    cd_convert_uoid2doid              Convert UOID to DOID
    cd_sync_object                    Get the server to send a newly
                                      replicated object
    cd_bg_init                        Initialize CD background
                                      processes
    cd_bg_merge                       Execute a background merge
    cd_bg_sync_remote_upto_ptid
                                      Bring remote location up to date
                                      with local PTID
    cdi_init
    cdi_shutdown
    cdi_execute_ack_sys               Execute acknowledged request
                                      using system session
    cdi_execute_ack                   Execute acknowledged request
    cdi_apply_locks                   Apply locks for txn
    cdi_abort_prc_txn                 Remove all locks already set for
                                      a txn
    //Forced update location (used to change update location
    when executing clash handler functions)
    cdi_register_forced_update_location
                                      Register location to be used as
                                      update location for thread
    cdi_unregister_forced_update_location
                                      Unregister location to be used as
                                      update location for thread
    cdi_get_forced_update_location
                                      Get forced update location for
                                      thread
    cdi_sync_upto_ptid                Bring location up to date with
                                      PTID
    cdi_sync_upto_now                 Bring location up to date with
                                      latest PTID
    cdi_sync_loc_list                 Make my location list consistent
                                      with destination location list
                                      and return info on mismatch of
                                      PTIDs
    cdi_read_loc_list                 Read location list
    cdi_sync_upto_dtid                Bring location up to date with
                                      DTID
```

Since updates are cached during a transaction, special handling of reads performed when updates are cached is required. In one embodiment, the caller of cd_read() or cd_readn() sees the results of all updates previously executed in the transaction. In an alternative embodiment, for cd_read() reads will see all previously added objects and will see the modified attributes of objects, but will not see the effects of moves or removes. Thus if an object is removed

cpu_abort_prc_txn	Remove object locks for specified transaction
cpsm_sync_upto_ptid	Bring remote locations up to date as far as given PTID
cpsm_get_latest_ptid	Obtain the latest PTID
cpsm_get_sync_object	Remote machine wants to sync a newly replicated object
cpsm_sync_object	Add a newly replicated object to the local database
cpsm_get_sync_update	Get a local sync_update
cpsm_sync_update	Apply multiple update txns to location
cpsm_read_loc_list	Read list of locations and states
cpsm_sync_loc_list	Sync location list
cpsm_merge_loc_list	Attempt to merge my location list with other location list
cpsm_sync_finished	Remote machine is notifying us that a sync_upto_ptid has completed
cpsm_request_merge	Request a merge of this location with the central server
cpui_init	Initialize internal structures
cpui_shutdown	Shutdown CPUI subsystem
cpui_execute_txn	Execute update txn at a local location
cpui_apply_update_list_to_db	Apply an update list to an OP database
cpui_commit	Commit all txns at location
cpui_flush	Flush all txns to object database at location
cpui_replay_logged_transactions	Replay transactions from the log that have not been committed to OP
cp_bg_init	Initialize CP_BG subsystem
cp_bg_shutdown	Shutdown CP_BG subsystem
cp_bg_handle_distributed_request	Handle a request that requires remote communication
cp_bg_notify_close_txn	Notify CP_BG of a closed transaction
cp_bg_notify_commit	Notify CP_BG that all txns are committed at a location
cp_bg_attempt_send_flush	Attempt to send out and flush txns
cp_bg_notify_load	Notify CP_BG of a newly loaded DB
cp_bg_notify_unload	Notify CP_BG of a newly unloaded DB
cp_bg_flush_upto_ptid	Force all transactions upto the specified ptid to the migrated state

The location distributor 78 in each replica manager 46 and the location state processor 80 are used to determine the storage locations of database entries. In one embodiment, the

The replica managers 46 track the last transaction sequence number made by every location up to the low watermark PTID in order to know whether a location is up to date with another location's low watermark. The log ordering may be different in different locations, up to an interleave.

One embodiment of the location state processor 80 provides functionality according to the following interface:

ls_init	Initialize LS
ls_shutdown	Shutdown LS
ls_close_db	Clear out all entries for a database
ls_allocate_new_lid	Allocate a new location identifier for use by a new replica
ls_add	Add a new location
ls_remove	Remove a location
ls_modify_local_tid	Modify a location entry's local transaction identifier (sequence number)
ls_modify_state	Modify a location entry's state
ls_get_loc_list	Get list of locations
ls_get_loc_sync_list	Get list of locations for syncing
ls_get_next_loc	Get next location
ls_get_first_in_loc_list	Get first location in list that is in current location set
ls_get_loc_entry	Get location entry given lid (location identifier)
ls_get_first_ref_loc	Get nearest reference location in provided list
ls_get_first_ref_loc_in_list	Get first reference location in provided list
ls_get_lock_loc	Get lock location for location set
ls_higher_priority	Determine which location has highest priority
ls_complete_merge	Complete the merge process
ls_set_sync_watermarks	Set the high and low watermark PTIDs used in syncing and merging

The object distributor 82 manages ACLs and otherwise manages access to objects in the database. In one embodiment, the object distributor 82 provides functionality according to this interface:

```
typedef void*  ndr_od_db_handle;   //open database handle
//lint -strong(AJX,ndr_od_txn_id)
```

not written to disk. NdrOdCommit() is used to commit closed updates to disk. However, after calling NdrOdCloseTxn(), no further updates may be applied in the transaction. This function is also where all the locking and updates previously cached actually get done. Consequently, most locking and/or consistency errors are reported here (after synchronization) so that the transaction can be retried:

```
ndr_ret EXPORT
NdrOdCloseTxn(ndr_od_txn_id        txn_id);    /* -> txn_id
*/
```

The NdrOdEndTxn() function ends the current transaction and executes an implicit NdrOdCloseTxn(). No error is returned if no transaction is currently open:

```
ndr_ret EXPORT
NdrOdEndTxn(ndr_od_txn_id        txn_id);    /* -> txn id    */
```

The NdrOdCommit function commits all previously closed transactions for the session to disk:

```
ndr_ret EXPORT
NdrOdCommit(
     ndr_od_db_handle         db,        /* -> DB to commit    */
     ndr_dodb_session_type    session); /* -> session  */
```

The interface also includes the following functions:

```
//Abort current txn
ndr_ret EXPORT
NdrOdAbortTxn(ndr_od_txn_id        txn_id);    /* -> txn_id
*/

//Get info on current txn
ndr_ret EXPORT
NdrOdGetTxnInfo(
     ndr_od_txn_id         txn_id,        /* -> txn_id   */
     ndr_od_txn_info*      txn_info);     /* <- txn_info */

//Lookup an object using parent Distributed Object Identifier
//(DOID; encodes location info to assist in sending distributor
//requests to the right machine; includes UOID) & sibling key
or
//using global key; the key value MUST be a contiguous
structure.
ndr_ret EXPORT
NdrOdLookupByKey(
     ndr_od_txn_id         txn_id,              /* -> txn_id   */
```

```
     /* <- Final length of data read  */
     ndr_os_object*         object);
     /* -> Pointer to object buffer   */

//Read an object using DOID
ndr_ret EXPORT
NdrOdRead(
     ndr_od_txn_id          txn_id,          /* -> txn_id   */
     ndr_dodb_access_rights_type rights_needed_on_parent,
     /* -> rights needed on parent */
     ndr_os_class           class_id,
     /* -> Class id. of superclass to match */
     /*    and superclass structure to be returned */
     ndr_dodb_doid_class* doid,          /* -> DOID  */
     UINT16                 max_length,
     /* -> Max length of data read */
     UINT16*                length,
     /* <- Final length of data read  */
     ndr_os_object*         object);
     /* -> Pointer to object buffer   */
```

An NdrOdReadn() function which reads multiple objects using parent DOID and wildcards behaves as if none of the updates in the transaction have been applied. Interpretation of wildcard values in the key is done by registered keying functions. NdrOdReadn() reads either up to max_objects, or up to the maximum number of objects that will fit in the max_length object buffer:

```
ndr_ret EXPORT
NdrOdReadn(
     ndr_od_txn_id          txn_id,          /* -> txn_id   */
     ndr_dodb_access_rights_type rights_needed_on_parent,
     /* -> rights needed on parent */
     ndr_os_class           class_id,
     /* -> Class id. of superclass to match
     and superclass structure to be returned */
     ndr_os_class         read_as_class,
     /* ->     Class id. target objects are to be read as   */
     ndr_dodb_doid_class* parent_doid,  /* -> Parent DOID */
     ndr_os_attribute       key_id,  /* -> Type of unique key */
     UINT16                 key_length,
     /* -> Length, in bytes, of the key value */
     VOID*               key,
     /* -> Key value to match, can contain wildcard.
     NULL implies match all objects under parent containing
     the key id */
     UINT16                 max_length,
     /* -> Max length of data read */
     UINT16*                length,
     /* <- Final length of data read  */
     ndr_dodb_object_list*   object_list,
```

```
//Remove agent defined lock
ndr_ret EXPORT
NdrOdRemoveAgentLock(
     ndr_od_txn_id           txn_id,  /* -> txn_id    */
     ndr_dodb_doid_class* doid,       /* -> Objects's DOID */
     ndr_dodb_lock_type         lock_type);
     /* -> Type of lock                    */
```

The following four calls are used to append various types of updates onto an open transaction. Any of them may return NDR_OK indicating success, NDR_CD_EXCEEDED_TXN_LIMITS indicating that transaction limits have been exceeded, or some other error indicator. In the case of exceeded transaction limits the transaction state will not have been changed and the failed call will have had no effect. The caller is expected to commit or abort the transaction as appropriate. In all other error cases the transaction is automatically aborted before returning the error to the caller:

```
//Modify a single attribute in a previously read object
//The object distributor caches the modifications and only
//applies them at close txn time
ndr_ret EXPORT
NdrOdModifyAttribute(
     ndr_od_txn_id          txn_id,        /* -> txn_id    */
     ndr_dodb_access_rights_type rights_needed_on_parent,
     /* -> rights needed on parent */
     ndr_dodb_doid_class* doid,
     /* ->    DOID of previous read version of object.
     Used to verify object has not been modified by another
     user since previously read */
     ndr_os_attribute      attribute_id,
     /* -> Identifies attribute to be modified    */
     VOID*                value);     /* -> New attribute value */

//Add a new object
//The DOID attribute does not need to be filled in by the
caller.
//The DOID will be set up before writing the object to the
//database.
ndr_ret EXPORT
NdrOdAdd(
     ndr_od_txn_id          txn_id,        /* -> txn_id    */
     ndr_dodb_access_rights_type rights_needed_on_parent,
     /* -> rights needed on parent */
     ndr_dodb_doid_class* parent_doid,  /* -> Parent DOID */
     ndr_os_class           class_id,
     /* ->   Class id of object */
     ndr_os_object*         object);
```

```
NdrOdGetAccessRights(
     ndr_od_txn_id          txn_id,        /* -> txn_id   */
     ndr_dodb_doid_class* doid,            /* -> Object DOID */
     UINT16*                acl_count,
     /* <- Number of ACL entries for that object */
     ndr_dodb_acl_element_type* acl);
     /* <- Rights information for that object   */

//Get the effective access rights for the current session
//for an object
ndr_ret EXPORT
NdrOdGetEffectiveAccessRight(
     ndr_od_txn_id               txn_id,    /* -> txn_id   */
     ndr_dodb_doid_class*        doid,      /* -> Object DOID */
     ndr_dodb_access_rights_type* rights);
     /* <- Effective rights for the current session  */

//Convert UOID to DOID
ndr_ret EXPORT
NdrOdConvertUoid2Doid(
     ndr_os_class                class_id,
     /* -> Class id. of object  */
     ndr_dodb_uoid_type*         uoid,      /* -> UOID   */
     ndr_dodb_doid_class*        doid);     /* <- Updated DOID */

//Convert UOID to DOID
ndr_ret EXPORT
NdrOdConvertUoid2LocalDoid(
     ndr_os_class                class_id,
     /* -> Class id. of object  */
     ndr_dodb_lid_type           location,
     /* -> Location on which object exists */
     ndr_dodb_uoid_type*         uoid,      /* -> UOID   */
     ndr_dodb_doid_class*        doid);     /* <- Updated DOID */
```

The object processor 86 provides a local hierarchical object-oriented database for objects whose syntax is defined in the object schema 84. In one embodiment, the object processor 86 is built as a layered structure providing functionality according to an interface in the structure which is described below. The embodiment also includes a module for object attribute semantics processing, a set of global secondary indexes, a hierarchy manager, a B-tree manager, a record manager, and a page manager. Suitable modules and managers are readily obtained or constructed by those familiar with database internals. A brief description of the various components follows.

op_dump_stats Dump statistics to the log

Due to higher level requirements of trigger functions in a set of trigger function registrations 94, in one embodiment it is necessary to have the old values of modified attributes available on a selective basis. This is done by means of a 'preservation list' produced by op_execute_updates(). The preservation list contains an update list specifying old attribute values for all executed updates that require it (as determined by a callback function), together with pointers to the original causative updates. These updates may not actually be present in the input update list, as in the case of an object removal that generates removes for any descendant objects it may have. Preservation lists reside in object processor 86 memory and must thus be freed up by the caller as soon as they are no longer needed.

The transaction logger 88 provides a generic transaction log subsystem. The logs maintained by the logger 88 provide keyed access to transaction updates keyed according to location identifier and processor transaction identifier (PTID). In one embodiment, a non-write-through cache is used to batch uncommitted transaction updates.

The transaction logger 88 is used by the consistency processor 76 to support fast recovery after a crash. Recovery causes the target database to be updated with any transactions that were committed to the log by the logger 88 but were not written to the target database. The log file header contains a "shutdown OK" flag which is used on startup to determine if recovery is required for the location.

replica 56 is modified and closed, the file distributors 90 and file processors 92 at the various locations holding the replicas 56 ensure that all replicas 56 of the file receive the new contents. It is not necessary for the agent to expressly manage any aspect of file content distribution.

A distributed file is identified by the UOID of the corresponding object; no built-in hierarchical naming scheme is used. A transaction identifier is also required when opening a file, to identify the session for which the file is to be opened. In one embodiment, the file distributor 90 and file processor 92 provide functionality according to the following interface:

```
//An ndr_fd_fork_id is the Id by which an FD open fork is known
typedef SINT16 ndr_fd_fork_id;
#define  NDR_FD_NOT_A_FORK_ID (-1)
//An ndr_fd_open_mode is a bit-mask which specifies whether a
//fork is open for reading and/or writing
typedef UINT16 ndr_fd_open_mode;
#define    NDR_FD_OPEN_READ_MODE           0x0001
#define    NDR_FD_OPEN_WRITE_MODE          0x0002
#define    NDR_FD_OPEN_EXCL_MODE           0x0004
#define    NDR_FD_OPEN_EXTERNAL_MODES      0x0007
//The remaining open modes are private to the replica managers
#define    NDR_FD_OPEN_SYNC_MODE           0x0008
#define    NDR_FD_OPEN_CLOSE_ON_EOF_MODE   0x0010
#define    NDR_FD_OPEN_READ_NOW            0x0020
```

In one alternative embodiment, opening a file with an NdrFdOpenFile() function returns pointers to two functions together with a separate fork_id for use with these two functions only. These pointers are of the type ndr_fd_io_function, and may be used as alternatives to NdrFdReadFile() and NdrFdWriteFile() when accessing that open file only. The functions should be at least as efficient as NdrFdReadFile() and NdrFdWriteFile() and will be significantly faster when the file access is to a local location. Their use does require that the caller maintain a mapping from the open

from the first location 36 to the replica 56 on the second location 38 is a state that is compatible with that replica 56.

By contrast, "persistent clashes" create inconsistencies that are present in the final states of two replicas 56. A clash whose type is unknown is a "potential clash."

A "file contents clash" occurs when a file's contents have been independently modified on two computers 28, or when a file has been removed from one replica 56 and the file's contents have been independently modified on another replica 56.

An "incompatible manipulation clash" occurs when an object's attributes have been independently modified, when an object has been removed in one replica 56 and the object's attributes have been modified in another replica 56, when an object has been removed in one replica 56 and moved in the hierarchy in another replica 56, when a parent object such as a file directory has been removed in one replica 56 and has been given a child object in another replica 56, or when an object has been independently moved in different ways. Thus, although clashes are discussed here in connection with files and the file distributor 90, clashes are not limited to updates involving files.

A "unique key clash" occurs when two different objects are given the same key and both objects reside in a portion of the database in which that key should be unique. In a database representing a file system hierarchy, for instance, operations that add, move, or modify files or directories may create a file or directory in one replica 56 that clashes on reconnection with a different but identically-named file or directory in another replica 56.

directory hierarchy" or "recovery directory" that contains a directory at the root of the volume, recovered items, and copies of any directories necessary to connect the recovered items properly with the root.

A clash handler function of one of the types below can be registered with the file distributor 90 for a database type to be called whenever the file distributor 90 detects a clash caused by disconnected modification or removal of a file's contents. The parameters are those of a regular clash handler plus the object DOID with NDR_OS_CLASS_FLAG_HAS_PARTIALLY_REPLICATED_FILE property (the file object defined by the object schema 84) and possibly a duplicated object return:

```
//Call back to a husk in respect of clashes detected at the
//database level
typedef  ndr_ret EXPORT (*ndr_fd_object_clash_fn)(
     ndr_od_db_handle             db,        /* -> Database */
     ndr_dodb_session_type        session,
     /* -> session to use in od_start_txn */
     ndr_od_clash_info*           info,
     /* -> Information on clash */
     ndr_dodb_doid_class*         old_doid,
     /* -> DOID of file with clashing contents     */
     ndr_dodb_doid_class*         new_doid);
     /* <- Doid of duplicated file                 */

//Call back to the husk in respect of clashes detected at the
//filesystem level
// (via pre trigger functions)
typedef  ndr_ret EXPORT (*ndr_fd_filesys_clash_fn)(
     ndr_od_db_handle             db,        /* -> Database */
     ndr_dodb_session_type        session,
     /* -> session to use in od_start_txn */
     ndr_od_clash_info*           info,
     /* -> Information on clash */
     ndr_dodb_doid_class*         doid);
     /* -> DOID of file with clashing contents     */
```

A parameter block such as the following is passed to clash handling functions to provide them with information about the clash:

```
typedef struct
```

to be called whenever the file distributor 90 creates a local copy of the file contents. This allows the replica manager 46 on a central server computer 28 to update the master copy of the file to reflect the attributes of the file created while disconnected:

```
typedef  ndr_ret EXPORT (*ndr_fd_creation_fn)(
     ndr_od_txn_id              txn_id,     /* -> txn_id   */
     ndr_os_class               class_id,
     /* -> Class ID of file */
     ndr_dodb_uoid_type*     uoid);        /* -> UOID of file */
```

The file distributor 90 embodiment also provides the following:

```
//Return aggregated information about all volumes
ndr_ret EXPORT
NdrFdVolumeInfo(
     ndr_od_txn_id          txn_id,          /* -> txn_id   */
     UINT32*                cluster_size,
     /* <- Number of bytes per cluster  */
     UINT16*                total_clusters,
     /* <- Total number of clusters     */
     UINT16*                free_clusters);
     /* <- Number of free clusters      */

//Add a file
ndr_ret EXPORT
NdrFdAddFile(
     ndr_od_txn_id          txn_id,      /* -> txn_id   */
     ndr_dodb_doid_class* doid,
     /* -> Uoid of file created */
     UINT32                 length);
     /* -> Length of existing file (0 when new) */

//Remove a file
ndr_ret EXPORT
NdrFdRemoveFile(
     ndr_od_txn_id         txn_id,      /* -> txn_id   */
     ndr_dodb_uoid_type* uoid);
     /* -> Uoid of file removed */

//Open a file for reading or writing by a task
ndr_ret EXPORT
NdrFdOpenFile(
     ndr_od_txn_id         txn_id,   /* -> txn_id   */
     ndr_os_class          class_id,
     /* -> Class ID of file to open */
     ndr_dodb_uoid_type  uoid,
     /* -> Uoid of file to open */
     ndr_fd_open_mode    open_mode,
     /* -> Open for read and/or write? */
     ndr_fd_fork_id*     fork_id,
```

```
        ndr_fd_fork_id          fork_id);       /* -> Id of open fork
*/

//Close a file, having completed reading and writing
ndr_ret EXPORT
NdrFdCloseFile(
        ndr_od_txn_id           txn_id,         /* -> txn_id    */
        ndr_fd_fork_id          fork_id);       /* -> Id of open fork
*/

//Given a UOID to a file or directory return its name
//in the specified namespace, along with its parent's UOID
ndr_ret    EXPORT
NdrFdGetFilename(
        ndr_od_db_handle             db,
        /* ->       handle to current database */
        ndr_dodb_uoid_type*          file_or_dir_id,
        /* ->       Uoid of object whose name is wanted */
        ndr_os_attr_property         namespace,
        /* ->       Namespace (e.g. DOS) of name wanted */
        void*                              name_buffer,
        /* <-       Buffer to receive name */
        UINT16*                            name_size,
        /* ->       Size of provided buffer */
        ndr_dodb_uoid_type*          parent_dir_id);
        /* <-       Parent UOID of object (NULL at root) */

//Callback functions to be used with
//NdrFdRegisterChangedIdCallback
typedef ndr_ret       EXPORT
(*NdrFdChangedIdCallback)(
        ndr_od_db_handle      db,           /* -> Database Id */
        ndr_os_class          class_id,
        /* -> Class ID of file or dir */
        ndr_dodb_uoid_type    *uoid,        /* -> Uoid of file or dir
*/
        UINT32                new_id);
        /* -> New Id allocated by underlying file system */
```

A NdrFdRegisterChangedIdCallback() function provides registration of a callback function to be called when a change to a file or directory's unique identifier is made. On a NetWare 4.x server this normally happens only when the file or directory is created by an internal file distributor 90 trigger function. However the identifier will be needed by agents for tasks such as directory enumeration. Because trigger functions cannot directly modify replicated objects, a record of the identifier change is queued within the file distributor 90 and the callback is made asynchronously:

```
//Synchronize all the files to and from this client for the
//passed database.  Return control when the files are up to
date.
ndr_ret EXPORT
NdrFdSynchronizeFiles(ndr_od_db_handle  db);

//Called from pre trigger functions to check whether
//or not the current connection has sufficient
//per-user-rights to perform a particular operation
//on a particular file system object.
ndr_ret
NdrFdCheckRights(
     ndr_dodb_uoid_type*          file_uoid,
     // uoid of object requiring rights to operation
     ndr_od_db_handle             db,
     // database raising the pre trigger
     UINT16                       operation);
     // bits representing operation

//Note that a file has been locally modified, setting
//modification info and triggering propagation onto other
//replicas.
ndr_ret EXPORT
NdrFdNoteFileModified(
     ndr_od_txn_id            txn_id,     /* -> txn_id   */
     ndr_dodb_doid_class*     file_doid);
```

The trigger function registrations 94 identify trigger functions that are provided by agents and registered with the object distributor 82. A registered trigger function is called on each event when the associated event occurs. Suitable events include object modification events such as the addition, removal, movement, or modification of an object. Because the trigger functions are called on each location, they can be used to handle mechanisms such as file replication, where the file contents are not stored within the target database, while ensuring that the existence, content, and location of the file tracks the modifications to the target database. All objects must have been locked, either implicitly or via NdrOdLock(), in the triggering transaction before the corresponding trigger function is called, and other objects may only be modified if the trigger function is being called for the first time at the location in question.

dispatch later receives the routed requests and dispatches them to an appropriate agent 44.

The agents 44, which have very little knowledge of the distributed nature of the database, invoke the consistency distributor 74, location distributor 78, object distributor 82, and/or file distributor 90. For example, a directory create would result in an object distributor 82 call to NdrOdAdd() to add a new object of type directory.

In contrast to the agents 44, the distributors 74,78,82, and 90 have little semantic knowledge of the data but know how it is distributed. The object distributor 82 uses the location distributor 78 to control multi-location operations such as replication and synchronization. The consistency distributor 74 manages transaction semantics, such as when it buffers updates made after a call to NdrOdStartTxn() and applies them atomically when NdrOdEndTxn() is called. The file distributor 90 manages the replication of file contents.

The processors 76, 86, 88, and 92 process requests for the local location 40. The consistency processor 76 handles transaction semantics and synchronization, and uses the transaction logger 88 to log updates to the database. The logged updates are used to synchronize other locations 40 and to provide recovery in the event of a clash or a crash. The logger 88 maintains a compressed transaction log. The log is "compressed," for example, in that multiple updates to the "last time modified" attribute of a file object will be represented by a single update. The logger 88 maintains a short sequential on-disk log of recent transactions; the

methods of the present invention. For instance, a file or directory may be renamed twice, rendering the first rename redundant. Likewise, a file may be modified twice, rendering the first update to the modification date redundant. Scripts or other mechanisms may also repeat operations to no further effect, such as deleting a file twice without recreating it in between the deletes or moving a file and then immediately returning it to its original location. These and similar redundant update sequences are identified during the step 100.

More complex but nonetheless redundant sequences can also be analyzed during the step 100. For instance, use of the location state processor 80 may identify an update in the transaction log that specifies an update location on a computer 40 other than the computer 40 which holds the log presently being managed. The log can then be compressed during the step 102 by removing that update.

In other situations, further steps are employed to identify redundant updates. For instance, a transaction identifying step 104 determines the most recent successfully merged transaction that updates a selected object. Transaction boundaries may be identified by checkpoints inserted in the log during transaction synchronization or by version control operations. Boundaries may be determined using the object processor 86 as described below in connection with certain three-level structures. Every transaction checkpoint is located at the boundary of a transaction, as defined by the three-level structures or other means, that is found in the log prior to compression. However, not every such boundary will be available as a checkpoint because compression may remove some

produce "add A; rename A Z; add B; add C" and subsequently managed by the consolidating step 110 to produce the sequence "add Z; add B; add C." Of course, other semantically equivalent sequences may also be produced according to the invention, such as the sequence "add B; add C; add Z."

In short, redundant updates are identified by examining the operations performed by the updates and the status of the replicas 56. Log compression is based on update semantics, unlike data compression methods such as run-length-encoding which are based only on data values.

During a creating step 112, a hierarchical log database is made. The log database represents at least a portion of the transaction log using objects which correspond to the updates and transactions in the specified portion of the transaction log. The log database is preferably efficiently cached and swapped to disk as needed.

In one embodiment, the log database is a hierarchical database maintained by the object processor 86. Transactions are represented as a three-level structure. A top-level transaction sequence object contains the PTID associated with the transaction that is described by the object's descendant objects. This PTID is a global key, with its sibling key being the log append order for the transaction in question.

An update container object, which is a child of the transaction sequence object, serves as the parent of the transaction's log database update objects. It is separated from the transaction sequence object in order to allow the updates in a transaction to migrate into another (by PTID) transaction during the repositioning step 108.

checkpoints. Each checkpoint attribute or checkpoint object contains the location identifier value(s) of the location(s) 40 to which the synchronization checkpoint pertains. In the case where the log is on a client 20, this will be the corresponding server 16. If no values are present, no synchronization has yet been done.

The portion of the transaction log represented by the log database may be the entire log, or it may be a smaller portion that only includes recent transactions. In the latter case, the remainder of the transaction log is represented by a compressed linearly accessed log stored on the device 54. In embodiments that do not include a log database, the entire transaction log is represented by a linearly accessed log stored on the device 54.

During one or more iterations of an inserting step 114 objects are inserted into the log database to represent updates, transactions, or synchronization checkpoints. Updates are represented as individual objects and determination of necessary management steps is often made at the update level.

However, the desired transactional treatment of updates requires that updates in a given transaction are always committed to the replica 56 together. Thus, in inserting an update as described herein, the replica manager 46 actually inserts a transaction containing that update. Likewise, in consolidating two updates from separate transactions, the replica manager 46 consolidates the transactions. And in moving an update, the replica manager 46 moves an entire transaction to make two transactions needing consolidation become adjacent to each other.

ii) Modifications to the object's parent or to the parent's naming; and

iii) Modifications to the object's old parent or to the old parent's naming (move-performing updates only).

Such separate tracking makes it possible to track naming changes (renames and moves) in order to identify incompressible sequences in step 106. Move operations are effectively naming changes in both source and parent directories, so move-performing updates go onto three chains. If object naming changes (through renames) were kept on the same chain as other object updates, then coupling of dependency chains through renames and moves could cause all dependencies to degenerate into one long chain which provides no benefit because it would be equivalent to linearly scanning the log in reverse order. Accordingly, separate tracking is utilized.

As noted, each completed transaction in the transaction log has a corresponding transaction sequence number. The transaction sequence numbers are consecutive and monotonic for all completed transactions. The transaction numbers are stored in transaction sequence objects in the log database. By specifying a range of one or more transaction sequence numbers, the replica manager 46 can retrieve transactions from the transaction log in order according to their respective transaction sequence numbers during a reading step 122.

One method of the present invention uses the transaction identifying step 104, the update history structure accessing step 116, and a constructing step 124 to provide a prior version of a target database object. More particularly, a list of attributes is constructed representing the attributes which

embodiment, functions are provided in the transaction logger 88 as follows:

object inserting step 114	tl_append() and/or tl_insert()
object removing step 120	tl_remove_record()
object modifying step 118	tl_remap_update_target(), tl_remap_uoid()
accessing step 116, identifying step 104, constructing step 124	tl_read_historical_object()
reading step 122	tl_readn()
identifying step 106	depend_update_dependent()
steps 108, 110, 102	various compression functions

In summary, the present invention provides a system and method for compressing a log of transactions performed on disconnectable computers. Redundant updates in the transaction log are identified through semantic tests and then removed. Operations are performed either directly on a disk-based log or by manipulation of objects and attributes in a log database. The present invention is well suited for use with systems and methods for transaction synchronization because the invention is implemented using replica managers 46 that perform synchronization and the log compression steps of the present invention may be used to remove transient clashes that arise during synchronization. The architecture of the present invention is not limited to file system operations but can instead be extended to support a variety of target and/or log database objects.

Although particular methods embodying the present invention are expressly illustrated and described herein, it will be appreciated that apparatus and article embodiments may be formed according to methods of the present invention. Unless otherwise expressly indicated, the description herein of methods of the present invention therefore extends to

CLAIMS

1. A method for managing a transaction log, the log representing a sequence of transactions in a network of connectable computers, each transaction containing at least one update targeting a target database object in a distributed hierarchical target database that contains convergently consistent replicas residing on separate computers in the network, said method comprising the computer-implemented steps of identifying at least one redundant update in the transaction log and then removing the redundant update from the transaction log.

2. The method of claim 1, further comprising the computer-implemented step of identifying an incompressible sequence of updates in the transaction log.

3. The method of claim 1, further comprising the computer-implemented step of identifying a transaction boundary within the transaction log.

4. The method of claim 3, wherein said step of removing the redundant update comprises the steps of determining the most recent successfully merged transaction that updates a selected object and then removing an update of the object that antedates the transaction.

5. The method of claim 1, wherein the transaction log resides on a first computer and said step of removing the redundant update comprises the steps of:

identifying an update in the transaction log that specifies an update location on a computer other than the first computer; and then

removing that update.

transaction to the transaction log by inserting a transaction object into the log database.

12. The method of claim 11, wherein said appending step comprises inserting an update object into the log database.

13. The method of claim 11, wherein said appending step comprises accessing an unreplicated attribute in the log database to identify an earlier update, if any, which references an object in the target database that is also referenced by an update in the appended transaction.

14. The method of claim 11, wherein said appending step comprises accessing an update history structure in the log database to identify an earlier update, if any, which references an object in the target database that is also referenced by an update in the appended transaction, the update history structure associating each target database object with the log database objects, if any, that correspond to updates referencing the given target database object.

15. The method of claim 8, wherein said method further comprises the computer-implemented step of adding a synchronization checkpoint to the transaction log by inserting a synchronization checkpoint object into the log database.

16. The method of claim 8, wherein said method further comprises the computer-implemented steps of removing a synchronization checkpoint from the transaction log by removing a synchronization checkpoint object from the log database and then compressing a previously incompressible region of the transaction log.

19. The method of claim 18, further comprising the computer-implemented steps of locating a transaction checkpoint, accessing the update history structure, and then constructing a prior version of a target database object.

20. A computer-readable storage medium having a configuration that represents data and instructions which cause a disconnectable computer to perform method steps for managing a transaction log, the log representing a sequence of transactions in a network of connectable computers, each transaction containing at least one update targeting a target database object in a distributed target database that contains convergently consistent replicas residing on separate computers in the network, the method comprising the computer-implemented steps of identifying at least one redundant update in the transaction log and then removing the redundant update from the transaction log.

21. The storage medium of claim 20, wherein the method further comprises the computer-implemented step of identifying an incompressible sequence of updates in the transaction log.

22. The storage medium of claim 20, wherein the method further comprises the computer-implemented step of identifying a transaction boundary within the transaction log.

23. The storage medium of claim 20, wherein the step of removing the redundant update comprises the steps of repositioning an update in the sequence of updates in the transaction log and then replacing the repositioned update and an adjacent update by a single equivalent update.

sequence numbers are consecutive and monotonic for all completed transactions.

30. A system for managing a transaction log, the log representing a sequence of transactions in a network of connectable computers, each transaction containing at least one update targeting a target database object in a distributed target database that contains convergently consistent replicas residing on separate computers in the network, said system comprising a computer comprising means for storing the log and means for executing programmed instructions, means for identifying at least one redundant update in the transaction log, and means for removing the redundant update from the transaction log.

31. The system of claim 30, further comprising means for identifying an incompressible sequence of updates in the transaction log.

32. The system of claim 30, further comprising means for identifying a transaction boundary within the transaction log.

33. The system of claim 30, further comprising means for creating a hierarchical log database representing at least a specified portion of the transaction log, the log database containing an object corresponding to an update in the specified portion and also containing an object corresponding to a transaction in the specified portion of the transaction log.

34. The system of claim 30, wherein said means for removing the redundant update comprises means for repositioning an update in the sequence of updates in the transaction log and

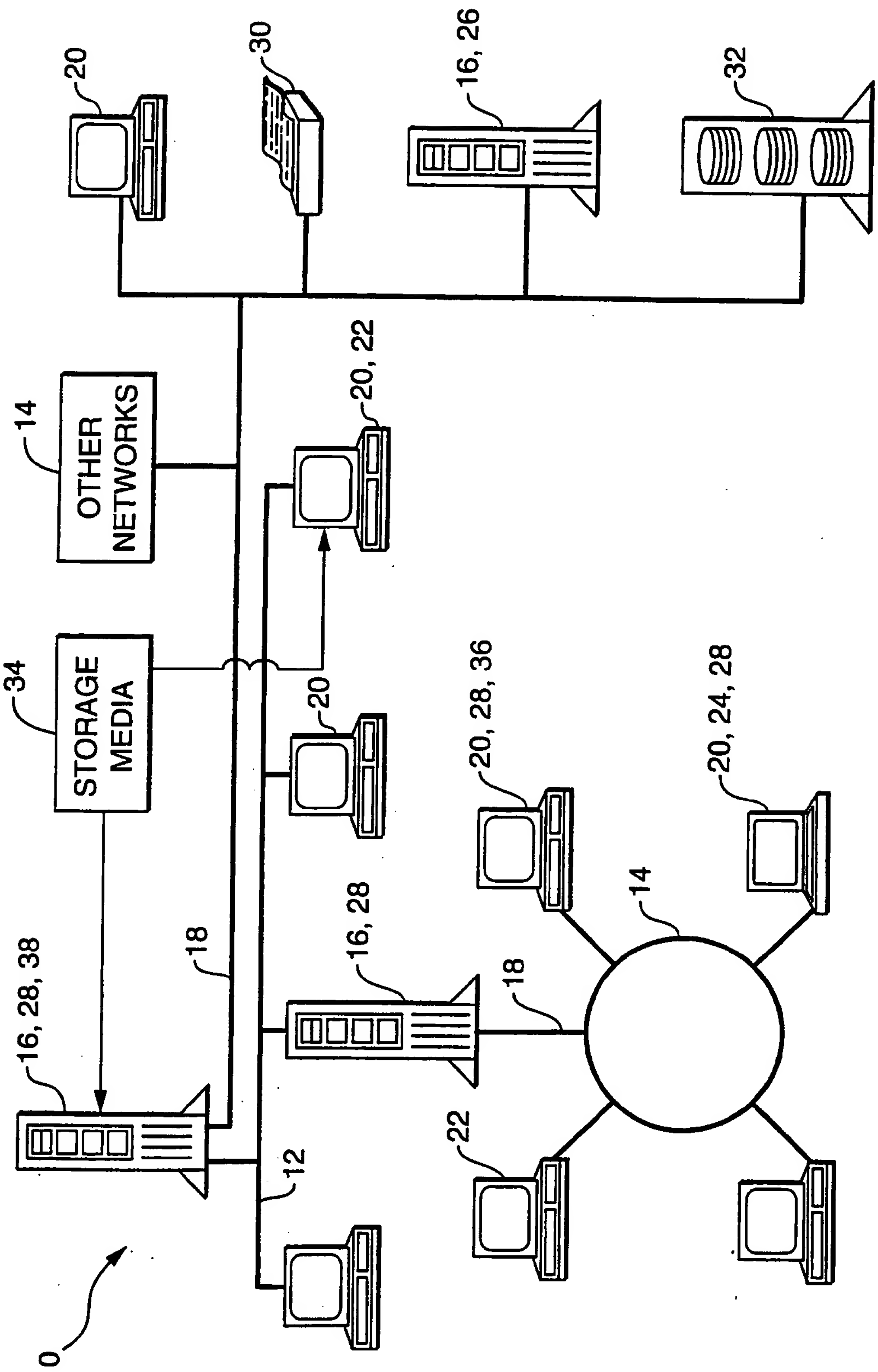

OTHER NETWORKS
STORAGE MEDIA
10
12
14
16, 28, 38
18
16, 28
20
20, 22
30
16, 26
32
34
20, 28, 36
20, 24, 28
22

FIG. 1

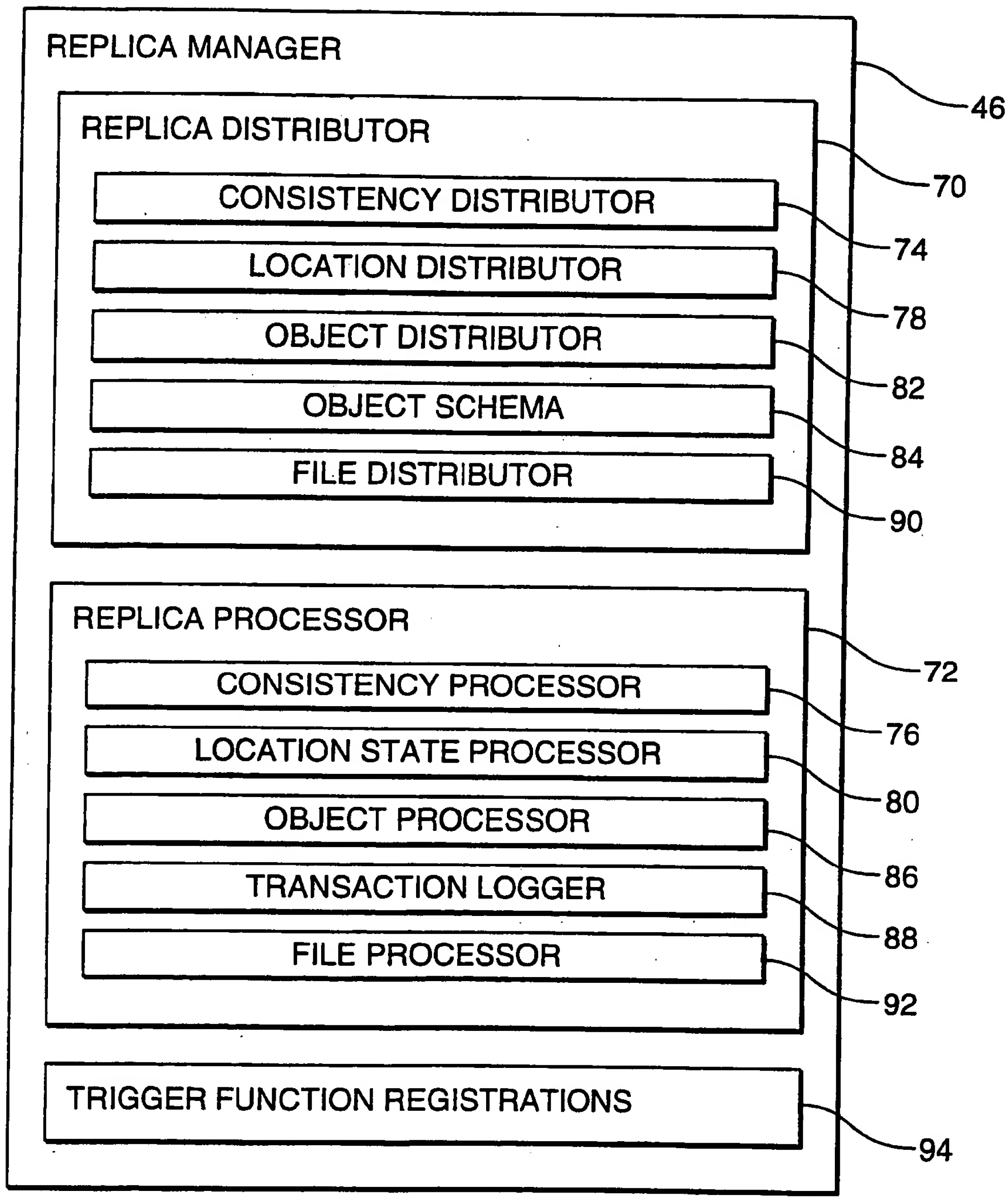

REPLICA MANAGER
REPLICA DISTRIBUTOR
CONSISTENCY DISTRIBUTOR
LOCATION DISTRIBUTOR
OBJECT DISTRIBUTOR
OBJECT SCHEMA
FILE DISTRIBUTOR
REPLICA PROCESSOR
CONSISTENCY PROCESSOR
LOCATION STATE PROCESSOR
OBJECT PROCESSOR
TRANSACTION LOGGER
FILE PROCESSOR
TRIGGER FUNCTION REGISTRATIONS
46
70
74
78
82
84
90
72
76
80
86
88
92
94

FIG. 3

INTERNATIONAL SEARCH REPORT

International Application No
PCT/US 96/11903

A. CLASSIFICATION OF SUBJECT MATTER
IPC 6 G06F11/14 G06F9/46 G06F17/30

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 6 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	C.J. DATE: "An Introduction to Database Systems, Volume II" July 1985 , ADDISON-WESLEY PUBLISHING COMPANY , READING, MA, US XP002016220	1,20,30
A	pages 1-33 (Chapter 1); pages 291-340 (Chapter 7) see page 13, line 1 - line 14 see page 291, line 1 - page 295, line 20 see page 306, line 34 - page 309, line 26 ---	2-6,18, 21-23, 31,32
A	EP,A,0 250 847 (IBM) 7 January 1988 see abstract see page 3, line 23 - page 4, line 10 see claim 1 ---	1,20,30

-/--

[X] Further documents are listed in the continuation of box C.

[X] Patent family members are listed in annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

1

Date of the actual completion of the international search	Date of mailing of the international search report
18 October 1996	04.11.96

Name and mailing address of the ISA	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	Wiltink, J

Form PCT/ISA/210 (second sheet) (July 1992)

INTERNATIONAL SEARCH REPORT

Information on patent family members

International Application No
PCT/US 96/11903

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
EP-A-0250847	07-01-88	US-A- 4878167	31-10-89
		DE-A- 3786956	16-09-93
		DE-T- 3786956	17-03-94
		JP-C- 1856908	07-07-94
		JP-A- 63010252	16-01-88

Form PCT/ISA/210 (patent family annex) (July 1992)